RUT - development manual 8.11 Testing of user interfaces v4.0-en

Gustav Nordvall

ABSTRACT

The user interface is a large part of most software on the market today. Since the user interface is the application's face towards the user, it's very important that it is appealing and that it works well. In order to assure this, the interface must be thoroughly tested. An important category of applications in this context is active web sites, which often consist of only one user interface and a database connection. An active web site runs in a browser and it's important to carefully test the user interface in different browsers because they may often yield different results.

The purpose of this document is to describe how the process of testing user interfaces can be planned and carried out.

1 Field of application

The field of application for this process description is all software development with a user interface. Test methods used to test user interfaces is described for the following four test phases: *the module test phase*, *the integration test phase*, *the system test phase* and *the acceptance test phase*. Since this process description also describes how testing should be planned it's recommended to read it in the definition phase or the design phase.

2 Prerequisites

Testing of the user interface can start as soon as there is a user interface module ready for testing, during the module test phase. When more user interface modules gets tested**,** their integration is ready to be tested in the integration test. Note that the implementation behind the functionality doesn't need to be finished to be able to test the user interface. Stubs are written for the missing functionality which for example gives static information to the user.

3 Realization

3.1 Overview

It's very important to find errors early in the program development. An appropriate test strategy is to divide the test into: module test phase, integration test phase, system test phase and acceptance test phase. See figure 1 for the different phases. How tests of the user interface might be planned and carried out in the different phases is described in this chapter. Other parts of the program are tested according to traditional methods, see [Bidarian, 1999]. The chapter finishes with a part consisting of active web sites and a brief description of software, on the market, which automates the work with testing.

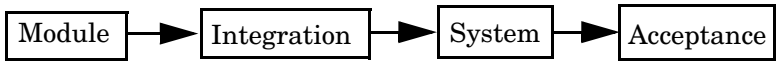

Figure 1: Test phases

3.2 Goals for user interface testing

As with all kinds of software testing, user interface testing strives to achieve certain goals. The main goals of this process are bug prevention, bug symptom discovery, and bug correction diagnoses [Beizer, 1990]. Prevention of all bugs in the user interface is naturally the ideal scenario. This saves both time and effort later in the project since no code needs to be corrected. Although, since complete bug prevention cannot be achieved, the testing process must yield results to reach the secondary goal of bug discovery [Beizer,

1990]. In order to accomplish this, the testing must be thorough and divided into small, detailed parts.

3.3 Detailed description - Module testing

Distribution

Even the user interface should be distributed in smaller separate parts, modules, which can be tested separate. These modules should coincide with the other module distribution of the program. A module can for example consist of a simple dialog box or parts of an application window. Parts that have a large functionality should be grouped into a separate module, for example graphic trees and tables.

Stubs

The implementation behind the presented functionality doesn't need to be finished to test the user interface. Missing functionality is replaced by stubs. Stubs can be written in two ways, they either return a static code according to a certain type, or they return "Not implemented!". Assume the application has a button "Start calculation". A stub for this button could for example be a dialog box with the text "Start calculating is not implemented yet". Writing stubs can be very time consuming, hence it's good to let the test group take part in the creation of the test-stubs which the implementation group uses in the implementation.

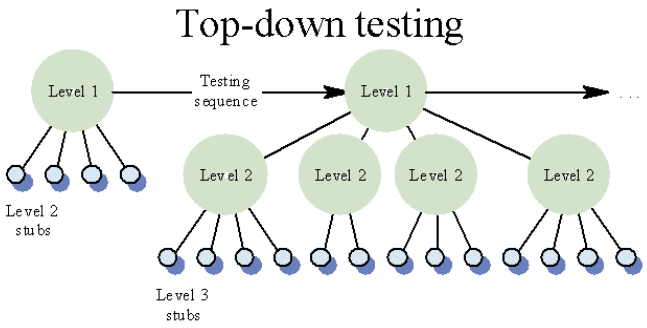

Figure 2: Testing according to the top-down model, using stubs.

Test cases and test scripts

Everything that will be tested is described with test cases and test scripts, exactly specifying what to be tested and the expected result. It should also be test cases and test scripts which test the basic condition. I.e. a windows appearance when it's first shown, the expected result could be compared with a low-fi prototype [Johannesson, 1998].

What is it you look for?

It's important to decide which types of errors you look for in every test phase. In this phase, it's important to look for misspellings and incorrect error handling. It should also be verified that the layout is appealing and functional. It's not enough to be satisfied with only clicking once on each button, that has probably already been tested by the implementation group. Instead, one should concentrate on testing incorrect error handling. Examples of what should be tested:

- What happens when input fields are left empty?
- What happens when you input very long strings?
- What happens if you input a decimal or a negative number?
- What happens if you click on buttons in the "wrong order"?
- Are error messages displayed when the user is doing something wrong?

Planning

Good planning is needed to for a satisfying module testing. To be able to test the modules in a convenient manner, the implementation order needs to be planned in advance. The implementation order should for that reason be included in the design specification. It's also good if it is decided in advance which parts that should be implemented as stubs before the module testing. If the work isn't planned carefully in advance, the implementation group will probably implement too big parts at a time and the test group will face an uneven workload.

3.4 Detailed description - Integration testing

Test against the underlying functionality

During the integration test, the integration between the modules of the system is to be tested. An appropriate strategy is to test the user interface against the underlying functionality. Practically, this means that the stubs used in the module test phase is replaced with real functions.

Test the modules against each other

When the modules of the user interface are tested with the underlying functionality, it's time to test the modules integration with each other, e.g. according to "Bottom-up" or "Top-down", see [Korpi, 1999] and [Rosengren, 1999] respectively.

Test cases and test scripts

Everything to be tested should be described with test cases and test scripts, which exactly specify what is to be tested and the expected result.

What is it you look for?

It's important to have decided what types of errors you are looking for in every test phase. Examples of what's supposed to be tested:

- Is the correct information presented?

- Is the information updated in the other modules when the information in one module is changed?
- What happens if you click on buttons in the “wrong order”?
- Are error messages displayed when the user does something wrong?

Planning

A good planning is needed to have the integration test working satisfactory. It’s important to have the module test phase performed in a pre-decided order so the integration test becomes a natural sequence of the modules which have already been tested.

3.5 Detailed description - System testing

Test the complete system

The system test tests the complete system. It’s not possible to test separate code functions any more. For that reason the system test only tests the user interface.

Test cases and test scripts

Everything that is to be tested should be described with test cases and test scripts, exactly specifying the expected result.

What is it you look for?

It’s important to have decided which types of errors you are looking for in every test phase. Examples of what is supposed to be tested:

- Are all requirements in the requirement specification document met?
- The performance should be tested for non-functional requirements.
- Does the user manual correspond with the implementation?

The system test should cover all functionality of the program. For that reason much of what was tested during the module testing and the integration testing might have to be tested again. The amount of test cases could be high, but if the module test and the integration test was performed carefully, there shouldn’t be many big errors left and the system test will become only a formal verifying.

3.6 Detailed description - Acceptance testing

Acceptance testing is similar to the system test (see 3.5), but is more focused on testing the usability of the system rather than compliance of the code against some specification [Van Vliet, 1996]. Usability testing includes letting a user with little or no prior knowledge of the system perform certain tasks (see 3.9). Acceptance testing is the major criteria for whether the system will be accepted or rejected by the customer.

3.7 Active web sites

Distribution

Every HTML file can be seen as a module. The HTML files are often tied up together with help of a “FRAMESET”. The stubs that are needed for the module test and the integration test are created by modifying the different HTML files to make them runnable in a separate browser. Many times, it could be a good idea to write a special “FRAMESET”, which only shows a subset of the application.

Different browsers

All testing should be done with different browsers of different versions since they interpret the code differently in some cases. The web sites, as a result of this, don’t look the same in different browsers. Some functionality also differ. This is extra problematic if Java Script is used. Every test case and test script could be used with all browser models. The test record is provided with separate columns for the different browsers which is ticked when the different browsers pass the tests.

Test against the underlying functionality

To perform module testing and integration testing of the parts not consisting of HTML code in an active web site is a little bit tricky since they often have to be executed from a browser. Even the modules which only execute in the web server often have to be started from a browser. The solution is to write small test programs in the form of simple HTML pages, calling the modules and presenting the result. The result can often be shown with the browser’s “View source” function. The other solution is to use the automated test tools discussed in the next chapter.

3.8 Automated test tools

There are tools on the market which automatically test the user interface. These programs can be instructed to execute interaction sequences, often done by the program “recording” user actions. Later these actions can be repeated over and over again. It usually works great to manually write test scripts which the test tool later can execute. After the test has been executed, the test tools have the ability to create test reports for the user to verify that the test is performed correctly.

With other words, test tools can perform module, integration and system testing. But they fail on the acceptance test (see 3.6) as this type of test requires users with little or no knowledge of the program being tested.

The big advantage with these types of tools is the simplicity to execute one and the same sequence of commands with different types of input and expected result. The tools should, however, only be seen as an aid. It’s still necessary to manually specify what is to be tested; you only gain the time to

execute the test. Time critical test cases may also be impossible to execute manually, which is something that these programs can handle superbly.

There are a lot of test tools available on the market today, what follows are descriptions of some of the automated test tools:

Vermont HighTest Plus

This is an automated software testing tool especially for the Microsoft Windows environment. The testing of web sites is done through Microsoft Internet Explorer, but this tool also handles standard windows applications as well.
HighTest Plus uses scripts to perform the testing. The scripts can be written by yourself using the provided script language or you could record your movements and interactions to create the script. The script language is made of only simple commands, such as Keys{“ENTER”,6.17} that will wait 6.17 seconds and then press the ENTER key on the keyboard.
When all test scripts are written to perform various test cases, they can be combined using HighTest's Interactive Suite Manager. That means that more than one test script can be executed at once, peforming the desired test.
To verify the results of a test, the Vermont HighTest Plus can compare screens, windows and files to see if the result is correct and the test is validated.
Also to take in consideration is the use of an integrated debugger. The main purpose of this function is to go line by line in the test script to examine the output and internal variables in the script in real time.
Moreover, a log file is created to make a detailed description of how the test did proceed.
With this type of the testing that the tool performs, it can easily be used to test modules in the interface and the integration with other modules.
See the Vermont web site [Vermont HighTest, 2006] for more information about this application.

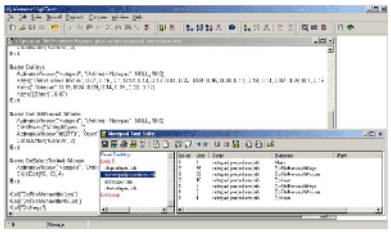

Figure 3: Screenshot of Vermont HighTest Plus.

QUOTIUM QTest

QUOTIUM QTest is an application that can perform so called “load testing” or “software stress”. This type of test is necessary when there are a great number of users using a web site. Basically, what the application does, is to make a heavy load on the web server and identify where the performance is critical. This type of test is difficult to perform by yourself as it requires a lot of resources. Therefore QTest creates more than 10000 “virtual users” that accesses the web site and there are possibilities to add and edit own test scripts that the “users” should perform.
The test scripts is written in a specific language called SCL to make a inter-

face between the browser and the web server. And also to create what QUOTIUM calls “control points” in the web sites, to test if a certain “control point” has been reached or not.
What QTest also offers are test scenarios, they can be created by automatic recording of how the user interacts with any browser available. Each of these test scenarios can then be validated, by having QTest performing them again. Also with the use of an integrated debugger if needed for more information.
After the tests have been performed, QTest gives the user feedback by graphs, summary list and detailed descriptions of errors identified.

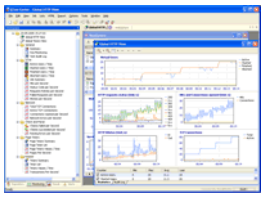

Figure 4: Screenshot of QUOTIUM *QTest.*

See the QUOTIUM web site [QUOTIUM QTest, 2006] for more information about this application.

TestWorks

The last example of test tools brought up is Software Research TestWorks. There exists a lot of tools, but not so many that could handle the UNIX platform. TestWorks exists in two versions, one for the Windows and one for the UNIX platform.
Like the other test tools, TestWorks captures the mouse movements, keyboard events and creates so called test scripts. The scripts are written in a own language, commands like “cb_mouse_button(“LEFT”,1102,473,129)” will press the left mouse button at the location 473, 129 on the screen after the time 1102 milliseconds.
TestWorks is divided into three parts, the first one, regression, to execute and records tests that is tolerant of smaller changes. Tolerant means that it contains advanced features to compare images of recorded material. The recording of tests extracts images that is compared to know if the test was a success or not. The test scripts can be combined as with the other mentioned tools and the results are generated and brought up to the user in a graphical way of trees and messages.
The next part is the coverage that can tell the user if code was executed or not, but does only works if the application was written in C, C++, Ada or FORTRAN. Also the last part, the advisor is a code tool to measure the logic and structure of the code to determine complexity.

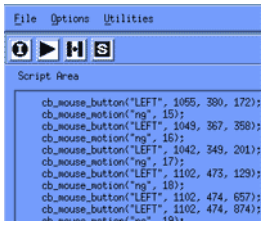

Figure 5: Screenshot of TestWorks.

See the Software Research Inc. web site [Software Research Inc., 2006] for more information about this application.

3.9 Usability testing

An alternative method of performing user interface tests is by implementing usability testing. Usability testing means measuring how well people can use some human-made object, such as a web site. The main purpose of usability testing is to discover difficultes, such as reading instructions, manipulatin parts or interpret the feedback people are given. If difficults are discovered, designers need to improve the interface and test it again. Also during the test which should be performed in as realistic a situtation as possible, the aim is to observe and discoverer areas of improvement. To simply gather opinions is not usability testing, you must arrange an experiment that measures a subject's ability to use the web site. [Wikipedia, 2006]

Performing a usability test involves creating a scenarios including specifed tasks that the user should perform. An example of a scenario of a web site, let's say "the Current Population Survey Site", could include tasks with questions, such as "What is the poverty rate for high school dropouts 25 years or older?" [Usability Testing of WWW sites, 2006]. The user of the web site would have to search for this information in the site and the observers write down time (the time it took to complete the task), accuracy (the number of misstakes), recall (how much the person did remember afterwards) and emotional respone (how the person feels about the task, stressed or confident). [Wikipedia, 2006]

Usability testing can be performed in different ways. Some examples are *structured usability tests*, *pair usability tests*, and *walk-up-and-use tests* [Monk et al., 1993].

In a structured usability test a user gets a list with a set of tasks which to solve by using the system. The user has to "think out loud" and describe how he or she reasons. The test is monitored by and observer, and is repeated several times with different users.

In pair testing two users work together to solve a given set of tasks. A test guide helps the users if they get stuck too long, and the test is also monitored by an observer.

In a walk-up-and-use test a user is presented with the user interface without any prior information or given set of tasks. The user defines the tasks by what he or she wants to do. This might make the testing evaluation harder to generalize, but is the most realistic usability testing method.

4 Results

4.1 Products

The result after this process is a basis for test planning and the providing of test cases, test scripts and test records. This will hopefully result in a user interface without errors (or at least with fewer errors).

4.2 Product templates

See chapter 5.

5 Templates and forms

Section 5.1 and section 5.2 are examples of how module test cases and module test scripts could look like for a log-on module in an active web site application.

5.1 Example templates of test cases

MT11	**Purpose**	To test the base condition.
	References	authentication.asp.
	Input	None.
	Performance	authentication.asp loaded with a browser.
	Expected results	The log-on page should be shown, consisting of a web page which greets the user, input-fields to fill in username and password and a “Log on” button.
MT12	**Purpose**	To test that it’s possible to log on.
	References	authentication.asp, main.asp.
	Input	Valid username and valid password.
	Performance	The user writes the input and clicks on “Log on”.
	Expected results	main.asp should be shown. If main.asp isn’t implemented, it’s enough to change the URL to main.asp.
MT13	**Purpose**	To test that an error message is received when loging on with wrong username or password.
	References	authentication.asp.
	Input	Invalid username and valid password**.** Valid username and invalid pasword.
	Performance	The user writes the input and clicks on “Log on”.
	Expected results	A dialog window with an error message should be shown.

Table 1: Test cases which test the authentication module.

5.2 Example templates of test scripts

MT11

1. Start Netscape Navigator 4.0 and Internet Explorer 4.0, if they aren’t already started.
2. Put in the address: http://xxx.ryd.student.liu.se/WebTRTest/authentication.asp into the browser’s address field.
3. The log-on page should appear consisting of a web site with a welcome greeting, an input field for a username and a password, and a “Log on” button.

MT12

1. Start Netscape Navigator 4.0 and Internet Explorer 4.0, if they aren’t already started.

2. Put in the address: http://xxx.ryd.student.liu.se/WebTRTest/authentication.asp into the browser's address field.
3. Put in username = "pum" and password = "pumma".
4. The site: http://xxx.ryd.student.liu.se/WebTRTest/main.asp should appear.

MT13

1. Start Netscape Navigator 4.0 and Internet Explorer 4.0, if they aren't already started.
2. Put in the address: http://xxx.ryd.student.liu.se/WebTRTest/authentication.asp into the browser's address field.
3. Put in username = "pummen" and password = "pumma".
4. A dialog window with an error message should appear.
5. Click on the "OK" button.
6. Make sure http://xxx.ryd.student.liu.se/WebTRTest/authentication.asp is shown. The username "pummen" should still be there.
7. Put in username = "pum" and password = "pumpum".
8. A dialog window with an error message should appear.
9. Click on the "OK" button.
10. Make sure http://xxx.ryd.student.liu.se/WebTRTest/authentication.asp is shown. The username "pum" should still be there and that the password input is cleared.

5.3 Example template of a test record

Module:			
Test responsible:		**Time consumption:**	
Test designer:		**Date:**	
Tester:		**Error report for test case number:**	
Test secretary:			

Test case				
No.		Test result	Comment	OK
	N			
	IE			

6 Verification of results

It's easy to verify the result directly in the test record, where expected results can be compared with the real result. The work with testing is not to be regarded as finished, until all tests have been completed with a correct result.

7 Examples with explanations

Section 5.1 and section 5.2 are examples of how module test cases and module test scripts could look like for a log-on module in an active web site application.

8 Solutions to common problems

A common problem with the module testing is when the implementation group and the test group have different opinions of the order the system should be implemented. If the work isn't planned correctly, the implementation group often wants to finish the system before they leave it for testing. The result of this is that the idea of module testing falls. For that reason, it's important to decide during the design phase in what order the modules should be implemented and what parts that should be implemented as stubs.

9 Adjustment to the PUM-course

This process description could be used without modification in the PUM-course project. In recent years, it's been common with active web sites in the projects; this process description is particulary preferred for that type of applications.

10 Measurement of the process

10.1 Resource measurement

The time consumed to plan and perform the test of the user interface is the measure of the resource for this process.

The time that's needed to test the user interface is strongly dependent of the complexity of the user interface. There is often a need to have many test cases to cover all functionality of the user interface. To write stubs for the module test is often a lot more time consuming than expected.

You have to be prepared to spend the same amount of time, or more, on testing the user interface as the time spent on implementing it.

10.2 Product measurement

A reasonable metric of the outcome of the testing is the amount of software errors found by the customer one year after delivery. A high amount of errors indicates that something went wrong during one or more test phases. To get an idea of what went wrong during what test phase it's a good idea to take notes of what types of errors that occur. Measured data is combined with help from the forms in section 10.3.

10.3 Forms for collection of data

There isn't any need for special resource measurement forms to keep a detailed time record of the work with coding.

Example template for collecting measured data

Date	Responsible	Description	Test phase	GUI

The example template above can be used for error reports of discovered errors after delivery. The phase when the error should have been discovered is indicated in the column "Test phase", to get information of which test phase that has to be improved. The column "GUI" indicates if the error was caused by the user interface or not with a simple YES or NO. The columns "Testphase" and "GUI" are filled in after the cause of the error has been investigated.

11 History of the process

Version	Date	Changes since last version	Author
1.0sv	2000-06-13	First version.	Tomas Franzon
2.0en	2004-01-06	Translated to English.	Stefan Hjelm
2.1en	2005-01-13	Correction of spelling and grammar.	Johan Sigholm
3.0en	2005-01-14	Some new sections added.	Johan Sigholm
4.0en	2006-01-06	Spelling and grammar corrections. Replaced the outdated automated test tools in chapter 3.8 with new tools with detailed descriptions. Added extensive explanation of usu-ability testing to chapter 3.9. Updated references according to the document. Diagram added to the overview. Modified changes to attend to.	Gustav Nordvall

12 Changes not yet attended to

- Evaluation of the methods needs to be improved.
- More detailed examples in chapter 7.
- Product templates in chapter 4.2 seems unnecessary.

13 References

13.1 Method description

[Beizer, 1990] Berizer, Boris, Software Tesing Techniques: Second Edition. International Thomson Computer Press. ISBN 1850328803.

[Bidarian, 1999] Bidarian, Arash, Val av teststrategi v5.0, RUT - Utvecklingshandbok 12.1.

[Johannesson, 1998] Johannesson, Thomas, Framtagning av användargränssnitt mha prototyper v1.3, RUT- Utvecklingshandbok 8.2.

[Korpi, 1999] Korpi, Mikael, Testgenomförande enligt Bottom up-metoden v4.0, RUT - Utvecklingshandbok 12.4.

[Monk et al., 1993] Monk, Andrew, et al., Improvingyour Human-Computer Interface. Prentice Hall, Hertfordshire. ISBN 0-13-010034-X (pbk).

[Rosengren, 1999] Rosengren, Fredrik, Top-down-testning v3.0, RUT - Utvecklingshandbok 12.3.

[Usability Testing of WWW Sites, 2006] http://www.bls.gov/ore/htm_papers/st960150.htm

[Van Vliet, 1996] Van Vliet, Hans, Software engineering principles and practice. John Wiley & Sons, USA. ISBN 0-471-93611-1.

[Wikipedia, 2006] http://en.wikipedia.org/wiki/Usability_testing

[QUOTIUM QTest, 2006] http://www.quotium.com

[Vermont HighTest Plus, 2006] http://www.vtsoft.com/vcsproducts/index.html

[Software Research Inc., 2006] http://www.soft.com/TestWorks/index.html

13.2 Method evaluation

[Myers, 1979] Myers, Glenford J, The Art of Software Testing, John Wiley & Sons, USA. ISBN 0-471-04328-1.

13.3 Internal comments

It's not trivial to find literature describing how user interfaces should be tested. You often have to be creative when it comes to test methods. It's not most important to follow a specific, accepted method. The main thing is to find errors early and get an application with as few errors as possible.